ChatGPT

Scraping Tab Audit Report

Detected Issues

- **Incomplete File Upload Handling:** The backend does not properly handle the CSV file upload. The frontend posts a file to the `/api/run` endpoint, but the server only defines a GET route for `/api/run` (no POST) to process an `input.csv`. This means the uploaded file is never actually saved or parsed, causing the scrape workflow to break.
- **UI Gaps & No Results Display:** The current UI only provides a file input and a "Start Scrape" button, with a simple status text. It doesn't display the scraped results in a table or any structured format. This diverges from the mockup design, which likely intended a results table and a more guided interface. The lack of a results view or download option is a significant gap.
- **Data Flow & Response Inconsistencies:** The frontend expects a JSON response (it calls `res.json()` on the fetch response), but the backend `/api/run` GET returns a plain text message ("Scrape complete") or a bare 500 error. This mismatch causes errors on the client (e.g. trying to parse a non-JSON response). Additionally, the code never informs the user of partial failures (like skipped rows), and uses a blocking request that can time out for large inputs.
- **CSV Validation Limitations:** The CSV input parsing only recognizes exactly two columns named "Keyword" and "Location" with values. It silently skips any row with an empty field or an unrecognized location. Locations must match a predefined city, ST format (only U.S. cities) – any variation (e.g. full state names or non-US locations) is considered invalid and dropped. There's no user feedback for these skipped entries or for completely invalid CSV headers, which can lead to confusion.
- **DataForSEO Polling & Logic Issues:** The integration with DataForSEO uses a task-based API for search volume and polls up to 3 times with a 2.5s interval. If DataForSEO hasn't returned results by then (which can happen for larger tasks or standard priority jobs), the code gives up and logs an error, returning zeros. This may prematurely mark the scrape "complete" with missing data. There is no batching of keywords (each keyword/location pair is posted individually), which is less efficient. Also, the code catches and suppresses errors by returning default values, potentially masking issues (e.g. invalid credentials or quota errors).
- **Styling & UX Divergence:** The implementation doesn't follow the project's UI/UX guidelines for consistency. For example, success and error messages are just plain text, not styled with the designated colors or badges (the style guide specifies using green/red text and background for statuses). The "Start Scrape" button styling (blue background) and input are not aligned with the primary design system (which uses indigo accents and specific Tailwind classes). The results are not shown in a formatted table, despite table styles being defined in the style guide. Overall, the scrape tab lacks the polished look and instructional text shown in the mockups (e.g. no indication of expected CSV format or sample download, no progress indicator for the scraping process).
- **Departures from Coding Rules:** There are minor divergences from the project's strict coding rules. Notably, a comment in the code suggests adding a Puppeteer-based scraper for content, which conflicts with the "no Puppeteer" rule. Regex usage is minimal (just for location parsing) and acceptable, but the location validation approach is rigid. There is also an implicit "file guessing" in using fixed filenames (`input.csv` and `results.csv`), though this is contained server-side. The

code adheres to ESM module format and doesn't rewrite entire files, so those rules are mostly respected. However, the incomplete file handling and use of a static CSV path indicate a need for a more robust approach in line with best practices (e.g. using a proper upload handler instead of assuming an `input.csv` will exist).

Component-Level Analysis

Frontend Scraper Component (Scraper.jsx & run.js)

The frontend `Scraper` page is very minimal. It provides a file picker and a button, then shows status text updates. Notably, it does not implement any display of the results after the scrape completes – it only shows a success message (`Scrape complete…`). This is a major functionality gap. In the intended design, after running a scrape, the user should see the output data (likely a table of keywords with their volumes, CPC, etc.). Currently, the React state does not even store the results – it only keeps `status`.

Another issue is how the frontend handles responses: it always calls `runScrape(file)` and then expects a JSON response containing a message. However, the backend is (in the current code) returning a plain string or no JSON at all on error. This leads to `err.message` like "Failed to start scrape: Internal Server Error" being shown to the user on failure – a technical message rather than a user-friendly error. The UI does not distinguish different error causes (e.g. invalid file vs server failure) since it just catches any exception in one block.

There's also no loading indicator beyond the button text changing to "Running…". For long-running scrapes, the user has no insight into progress – the design might have envisioned either a spinner or a progress bar, especially if multiple keyword-location pairs are processed. The UI resets the status when a new file is selected (which is good), but it doesn't clear the previous results because none are displayed in the first place. After a scrape, the file input remains populated, so pressing "Start Scrape" again will attempt the same file (which could be useful for re-runs, but the UI doesn't make it clear).

Stylistically, the frontend could be better integrated into the app's look-and-feel. According to the style guide, primary actions should use the indigo color scheme and rounded-2xl buttons, and status text should utilize the success/error styles instead of just an emoji prefix. The file input is a bare browser control; in the mockup, this might have been designed as a styled upload component or with an explanatory placeholder. These inconsistencies make the scraping tab feel less polished compared to other parts of the tool.

Backend Scrape Route (`backend/api/run.js`)

On the backend, the `run` route is where most issues lie. Currently, it only defines `router.get('/')` logic that expects an `input.csv` to be present on disk. There is **no `router.post` to accept the uploaded file** – a clear oversight in the implementation. In effect, unless the CSV is manually placed at the expected path, the GET handler will always fail to find `input.csv`. The correct flow should be: user uploads CSV -> server receives file -> server processes it and returns results. Right now, the code is split such that the processing is in GET, separate from the upload. It appears the developer may have tested by manually dropping a file named `input.csv` in the project folder and then calling the GET endpoint (which reads that file, processes keywords, writes `results.csv`, and returns a message). This workaround would not occur when using the UI, since the UI posts the file and expects a response in one go.

Within the processing logic, the route reads the CSV using `csv-parser`. It assumes the CSV has columns named exactly `Keyword` and `Location`. Each row is trimmed and validated. The location validation uses `normalizeLocation()` and `isValidLocation()` against a preloaded list of US city/state pairs. If a location doesn't match (e.g. "Los Angeles, CA" is fine, but "Los Angeles, California" or "Toronto, ON" would be rejected), the row is skipped with a console warning. The code does not collect these warnings to return to the user – so the user would only see fewer results than expected with no explanation. If *all* rows are filtered out (e.g. wrong headers or all invalid locations), `validPairs` ends up empty and the route responds with a 500 error "No keyword metrics retrieved." This is sent as plain text with a 500 status, which, as noted, front-end treats as a generic error. There's no explicit handling of CSV format errors – the code will simply produce zero `validPairs` if headers don't match ("Keyword"/"Location"), resulting in the same "No metrics" error.

After validating, the route iterates through each keyword/location and calls `getKeywordMetrics` (from the DataForSEO provider). This is done sequentially inside a for-loop. Each result is pushed into an array, and after all are done, it writes out a `results.csv` file. The CSV writing is straightforward (it uses the first result's keys as headers and writes every row). One issue here is the file path: it writes to a fixed path in the `backend/` directory (`results.csv`). If two scrapes run concurrently (even by the same user in quick succession), they would overwrite this file or conflict – the implementation is not multi-user safe. Even for single-user usage, having a static output file could be problematic if the user runs the tool multiple times (the file gets replaced each time). There's also no cleanup of `input.csv`; the code expects it to be overwritten on each new run (which would have happened if the POST saved it, but that part is missing).

Importantly, the route's response is currently `res.send('Scrape complete')` on success – with status 200 but no JSON structure. To adhere to the front-end expectations, it should return JSON (e.g. `{ message: "Scrape complete." }`). Similarly, error cases should send JSON error messages or at least set a clear status code that the front-end can interpret. The lack of a proper JSON response is a data flow error. Also, because the processing is done synchronously within the request, the client is left waiting for the entire duration of the scrape. For a large CSV (say hundreds of keywords), this could take tens of seconds or more. There is no streaming or asynchronous job handling – the simplest fix is to keep it synchronous but possibly this could block the Node event loop if not careful (each DataForSEO call is awaited serially).

In summary, the backend scrape route has broken logic (no file upload handling), and where it does work, it's limited to very specific input formats and will not inform the client of what happened beyond a generic success or failure. It also diverges from the roadmap by not supporting any advanced features that might have been envisioned (e.g. different scrape types or multi-step processing), and it violates the "no file guessing" principle by relying on hardcoded file paths rather than handling the uploaded file stream.

DataForSEO Integration & Polling

On the positive side, the tool uses DataForSEO's Keyword Data API to retrieve search volume, CPC, and competition – which aligns with project goals (getting keyword metrics at scale). This choice is appropriate given DataForSEO's reputation for accuracy and scalability in SEO data [1] [2]. The integration uses basic

auth with credentials from `.env`. However, the implementation has room for improvement in efficiency and reliability:

- **Sequential vs Bulk Tasks:** The code posts one keyword at a time to `.../search_volume/task_post`. DataForSEO actually allows batching multiple keywords in one request (by sending an array of tasks). By not leveraging this, the current approach incurs overhead for each keyword and is slower. Ideally, the backend could post all keywords in one batch call or in a few larger batches, then poll a combined result. This would also reduce the number of HTTP calls. The current design handles each keyword separately likely for simplicity, but it's not taking full advantage of DataForSEO's bulk support (which is noted as a feature of such APIs [3]).

- **Polling Logic:** The script waits up to ~7.5 seconds for each keyword's data, checking 3 times. Many times this will be sufficient (DataForSEO's standard queue often returns volume data quickly), but it's not guaranteed. Since the endpoints are asynchronous, a safer approach is either to poll more times (or with exponential backoff) or use DataForSEO's "tasks_ready" mechanism to retrieve results when done. Currently, if the data isn't ready after 3 attempts, `getKeywordMetrics` throws an error that is caught internally and returns `{ volume:0, cpc:0, competition:0 }` – effectively marking that keyword as having no data. The user would see a volume of 0 which might be incorrect (it could simply be that the result wasn't ready in time). This could mislead decisions. There's no notification to the user that some data might be missing due to timing out.

- **Error Handling:** The `getKeywordMetrics` function catches any error (including HTTP errors from the API) and logs it, then returns default 0-values. That means even if the API fails (due to invalid credentials, quota exceeded, or a network issue), the code would just produce zeros for all keywords rather than stopping or reporting an error. This is dangerous because it fails silently – the user might think those keywords truly have zero volume. For instance, if the DataForSEO account ran out of credits, every query would come back 0 without any explicit error message. According to DataForSEO's documentation, tasks return status codes and messages; the implementation should check for those and handle them properly instead of blindly treating missing data as zero.

- **No Caching of Results:** Each run will call the API for every keyword, even if some were recently looked up. Given that DataForSEO charges credits per API call, not caching results is a missed opportunity. For example, if the user uploads the same CSV twice, the second run could reuse the `results.csv` or an in-memory cache instead of querying again. The current code does not implement any caching or reuse (beyond writing `results.csv`, which it doesn't read back except via the separate results endpoint). Caching wasn't strictly required by the prompt, but it's mentioned as a best practice to consider. Implementing at least a simple cache (in-memory or on-disk) for keyword-volume lookups during a single session could optimize credit usage.

On the bright side, the integration is fully ESM and doesn't use any disallowed libraries. It avoids Puppeteer for scraping (adhering to the "no Puppeteer" rule) by relying on the DataForSEO API, which provides the needed data without HTML scraping. This is in line with the project guidelines and the known benefits of DataForSEO (providing real Google Ads data via API and being developer-friendly) [1]. The code also demonstrates correct use of async/await for API calls. Once the issues above are addressed, the DataForSEO integration can be a robust backbone for the scraping tool.

Results Storage & Presentation

After a scrape, results are saved to `results.csv` on the server. There is also a `/api/results` endpoint (in `api/results.js`) that reads this CSV and returns JSON data (using PapaParse). However, the frontend does not call this endpoint at all in the current implementation. This indicates an unfinished feature – likely the intention was to have the client fetch `/api/results` after the scrape completes, and then display the data. Because this call is never made, the parsed JSON is never delivered to the frontend. The presence of the `results` API route is good (it means the backend can serve the data in a structured way), but without frontend integration it's dormant.

Moreover, even if the frontend did call `/api/results`, there's a subtle issue: the `results.csv` path is assumed to be in the server's working directory. Depending on how the server is started, this could resolve to different locations (though in this project it's likely the root of the backend folder). They use `path.resolve('results.csv')` without a specific directory, which could be fine if the process CWD is consistent. It's a minor point, but it could fail if the working directory isn't as expected. A safer approach would be to store results in a known subfolder (like the existing `backend/uploads/` or a new `backend/output/`).

From a design perspective, the results are meant to be shown to the user in an understandable format. The likely plan (per the mockups and roadmap) was to present the CSV data in a table within the app, and possibly allow the user to download the CSV. Currently, the user has no indication where the results went – they don't see them, and unless they open dev tools or check the server, they wouldn't know a `results.csv` exists. This is a significant divergence from the expected user flow. In the mock "Scrape Builder" UI, we would anticipate a table listing each keyword with its volume, CPC, competition, etc., styled according to the app's tables style. The style guide explicitly provides table styling guidelines (full width, striped rows, etc.), implying that a results table was indeed intended. Not implementing this leaves the feature half-finished.

Additionally, the UX could be enhanced by offering a direct download of the results CSV (for example, a "Download Results" button once complete). Given that the data is already in a CSV file, it would be trivial to expose it via a static route or prompt a download. This isn't done yet, but would be a natural extension.

In summary, the back-and-forth flow of data (upload CSV -> process -> produce output -> show output) is broken after the processing step. The user is never shown the output. Fixing this is critical to meet the roadmap's goal of providing structured responses to the user.

Alignment with Roadmap & Mockups

Comparing the implementation to the project roadmap and mockups, several deviations are apparent:

- The **mockup for the "Scrape Builder" tab** likely includes a more user-friendly interface with instructions. For example, it might have indicated "Upload a CSV of keywords and locations to get search volumes" with perhaps a template link. The current UI simply has a header and an icon (🗔) with no further guidance. This is a UX gap. Incorporating helper text or a link to a sample CSV (as often seen in such tools) would align the tool with the roadmap's emphasis on ease-of-use.

- **Structured display of results** was expected (the roadmap mentions "structured response display"). The lack of any table or visualization means the implementation diverged significantly here. In the mockups, after running a scrape, the user should probably see a table or some analysis (maybe highlighting which keywords have good volume or competition). The code doesn't attempt any such analysis or scoring on the frontend. There was mention of an "auditAndScore" utility in commit notes, suggesting that the scrape might have been tied into a scoring mechanism to rate niches or opportunities. If that was on the roadmap, it's not reflected in the current code – no scoring of keywords is done apart from retrieving raw metrics. This is a divergence from the planned functionality (the tool is currently just a raw data fetcher, not a full "niche scout" or scoring tool as the name *MarketScout* might imply).

- The **coding guidelines** set forth (no Puppeteer, minimal regex, etc.) are mostly respected in implementation, but one point on the roadmap was likely to avoid hacks or quick fixes. The current file handling (writing to a fixed CSV, not handling concurrent usage, etc.) feels like a shortcut rather than a robust solution. It doesn't strictly violate a specific rule, but it's a technical divergence from best practices the project aspires to (e.g. the project likely expects proper use of libraries for file upload and not assuming global state). Also, the presence of a Puppeteer-related comment (placeholder) is concerning – it suggests a plan to use Puppeteer which should be reevaluated in light of the "no Puppeteer" rule. Instead, any required web scraping should go through allowed APIs (DataForSEO's SERP API, etc.) or other approved methods.

- The **UI consistency** with the rest of the Supertool is lacking. The roadmap and style guide emphasize a unified look. For instance, the sidebar and navigation tabs (as in other sections like Dashboard, Site Builder, etc.) should be present and highlight the current "Scrape" tab. If the Scraper page is not integrated into the navigation (it appears to be a standalone page component), it might be missing the surrounding layout that other pages have. This could be a simple matter of the codebase architecture (maybe they include the nav in a parent component and Scraper is just the inner content), but it's worth ensuring the Scrape tab is fully integrated into the app's navigation with the correct active states, icons, and so on.

Overall, the current state of the scraping tab falls short of the envisioned feature in the roadmap. Key functionalities (like parsing the input, polling results, and displaying them nicely to the user) are broken or incomplete. The next section outlines how to fix these issues and realign the implementation with the project's requirements.

Recommended Fixes

To address the issues above, we propose the following fixes and improvements, organized by area:

1. **Implement Proper File Upload Handling:** Introduce a POST route for `/api/run` that uses a middleware (e.g. multer) to handle multipart form-data. This route should accept the uploaded CSV file (field name "file"), validate its presence, and save it to the server (or parse it in-memory). For example, use `upload.single('file')` as in the existing upload routes, and write the file to the expected `input.csv` path or directly stream it into the CSV parser. This will eliminate the need for the client to do anything special – the single POST will handle upload and processing. Ensure this new code is added as a small change (respecting "no full file rewrites") – we can refactor the current GET handler's logic into a reusable function that the POST handler calls after saving the file. This

change closes the logic gap where the file wasn't being read, and adheres to the no-guessing rule by explicitly handling the file instead of assuming a filename.

2. **Return JSON Responses:** Modify the `/api/run` route responses to use JSON format consistently. On success, respond with a JSON object (e.g. `{ message: "Scrape complete", count: N }` where N is number of results or similar). On errors (invalid file, no valid rows, DataForSEO failure), send a JSON error with an explanatory message and an appropriate HTTP status (400 for bad input, 500 for server errors). The frontend's `runScrape` function can then remain largely the same, but we should adjust it to handle error responses more gracefully. For instance, if `res.ok` is false and the server sent JSON `{ error: "Message" }`, we can read that and show it. This way, the user will see a friendly error like " No valid keywords found in CSV" instead of a generic status text. Aligning the request/response format will make the data flow robust.

3. **Front-End: Fetch and Display Results:** After initiating the scrape, the frontend should retrieve and render the results. The simplest approach: once the POST `/api/run` responds with a success message, trigger a GET request to `/api/results` to fetch the parsed data. Use React state to store the returned data array. Then, add a component (or extend `Scraper.jsx`) to display this data in a table. Follow the style guide's table styling for a polished look (full width table, header with gray background and uppercase text, alternating row colors, etc.). Columns would be Keyword, Location, Search Volume, CPC, Competition – using the keys from the results JSON. If the roadmap/mocks intended additional computed metrics (like a "Score" or indicator of opportunity), those can be added if available (or left blank for now). By presenting the data in-app, we fulfill the "structured response display" requirement.

4. **Improve Status Feedback in UI:** Enhance the `status` messages and add loading indicators. For example, while the scrape is running, instead of just changing the button text, we can show a spinner icon next to "Running..." and maybe a note like "Processing X keywords...". After completion, if results are displayed in a table, we might change the status to " Completed. Showing results for N keywords." or similar, in a green styled text. Use Tailwind utility classes from the style guide for success and error states (e.g. `text-green-600 bg-green-50 p-2 rounded` for a success message badge, as suggested in the style guide). If any rows were skipped due to validation, we could also surface a warning (e.g. "⚠ 2 locations were unrecognized and were skipped") in a smaller, yellow-toned message. This requires tracking the skips in the backend and including that info in the response (we could easily count how many rows were dropped and send it in the JSON). Providing this feedback will make the tool more user-friendly and closer to the planned UX.

5. **Robust CSV Validation & Messaging:** On the backend, update the CSV parsing to be more flexible and informative. Possible improvements: allow case-insensitive header matching (e.g. "keyword" or "Keyword" both work), or at least detect missing expected columns and throw a specific error. If the user provided an unexpected format, return a 400 with a message like "CSV must include 'Keyword' and 'Location' columns.". When normalizing locations, consider common variations – perhaps accept full state names by mapping them to abbreviations (since the city list likely uses state codes). If a location is invalid, instead of silently skipping, collect these into an array. After processing, if any were skipped, include a note in the response (as mentioned above). These changes ensure the user isn't left guessing why certain entries didn't appear. All regex usage here is simple and necessary for parsing location format, so it remains within the "no regex unless necessary" rule (we aren't introducing any heavy regex processing, just a tweak to make it case-insensitive or to handle a wider

pattern if needed). Logging of validation issues can remain (for developers), but crucially, some of that info should bubble up to the user as well.

6. **Optimize DataForSEO Task Handling:** While maintaining the current DataForSEO integration (since it's working and aligned with the project's data source decisions), we can make it more efficient and resilient. In the short term, increase the polling attempts or interval for search volume data if needed – e.g. try 5 attempts over ~15 seconds for each keyword, or detect the `status_code` in the API response to decide whether to keep polling. DataForSEO's API might return a specific status code (e.g. 202 or a JSON field indicating "Pending"). Incorporate that so we don't count it as a failure prematurely. In the longer term, implement batching: if a CSV has many keywords, group them (for example, DataForSEO allows sending up to 1000 keywords in one POST). The response can then be fetched for all of them in one go (using their `/tasks_ready` or by checking each task ID in parallel). This would significantly speed up processing and reduce the chance of hitting the polling timeout for larger jobs. It will also adhere to DataForSEO's intended usage for bulk data [3]. We should also handle errors from the API more explicitly – if a task returns an error or if the credentials are bad, propagate a failure rather than treating it as zero volume. Perhaps the first sign of a global issue (like auth failure) should abort the whole scrape with an error status, so the user knows something is wrong (and can check API keys or credits). By doing this, we ensure the tool doesn't give misleading output. All these adjustments stay within the current DataForSEO integration (we're not replacing it with another service, just using it smarter), so they respect the project scope. Keep in mind to remain within the credit budget – if implementing re-polls or batch calls, avoid infinite loops or overly frequent polling that could waste credits. A balanced approach would be poll a bit longer, but eventually fail with an error message like "DataForSEO is taking too long – please try again" rather than silently timing out.

7. **Introduce Caching (if feasible):** In line with best practices, consider caching query results to avoid duplicate API calls. For example, maintain an in-memory cache (or a simple JSON file cache) keyed by `keyword+location` that stores the volume/CPC results for a certain duration (maybe during a single session or for a few hours). Before calling `getKeywordMetrics`, check the cache – if we have recent data for that pair, use it instead of calling the API again. This would be particularly useful if the user is experimenting with similar CSV inputs or rerunning the same file. Implementing this must be done carefully to not break the CSV workflow; a cache hit should still be recorded in the results and shown to the user transparently. Logging can note when a result was served from cache (to differentiate from a fresh API call, useful for debugging). This suggestion aligns with "caching best practices" and does not violate any rules (it reduces external calls and doesn't involve any disallowed operations). If caching is beyond the immediate scope, it can be a next-step enhancement, but it's worth planning for given the pay-per-use API model.

8. **UI/UX Enhancements per Design:** Refine the scraping tab's frontend to match the mockups and style guide. This includes styling the upload input (for example, using a styled `<label>` that triggers the file dialog, showing the selected file name), using the primary button style for "Start Scrape" (Indigo color, rounded corners, shadow, etc.), and ensuring the overall layout matches the rest of the app. The content could be wrapped in a card or container consistent with other pages (as indicated in the style guide for cards and spacing). Also, integrate the Scrape tab into the sidebar navigation if not already – the sidebar should highlight this tab when active, using the active tab style (indigo text and border). Essentially, the Scrape page should not feel like an isolated tool but rather one mode of the Supertool. Adding the missing explanatory text from the mockup is

important: e.g. a brief description above the file input like "Upload a CSV with columns 'Keyword' and 'Location' to fetch search volumes and CPC. You can get up to X keywords at once." This guides the user and reflects the roadmap's intent to make the tool intuitive. All these changes are purely frontend and won't affect the backend logic, but they bring the implementation in line with the project's polished UI standards.

9. **Remove/Replace Puppeteer Placeholder:** Since the coding rules forbid Puppeteer, any plan to use it for scraping should be abandoned. In the codebase, there is a placeholder for adding Puppeteer in a content fetching step (likely intended for getting on-page data or something). This should be replaced with approved methods – for example, DataForSEO's SERP API or other provider if the goal was to scrape SERP features or competitor content. Given the scope of the scraping tab as implemented is just keyword metrics, we might not need Puppeteer at all (and indeed, none is currently used). So ensure that placeholder comment is removed or clarified, to avoid future developers accidentally introducing Puppeteer in violation of the project rules. If additional scraping (like getting titles from competitor websites) is required by the roadmap, use the DataForSEO On-Page or SERP endpoints, or at least obtain approval for any headless browser usage if absolutely necessary. Keeping the codebase strictly Puppeteer-free is the safer route – it's in the rules for presumably good reasons (maintenance, cost, complexity), and our data needs are being met by APIs.

10. **Testing & Verification:** After implementing the above fixes, thoroughly test the end-to-end flow with various CSV inputs:

    - A valid small CSV (to verify basic functionality and UI display).
    - Edge cases like an empty CSV, missing headers, invalid locations (expect clear error messages).
    - A larger CSV with ~50-100 entries to observe performance and ensure the UI doesn't freeze and the polling adjusts properly (if it takes a bit longer, the user should still get the result without error).
    - Concurrent runs (if the application could allow it – e.g. two users or two different CSVs in short succession) to ensure that caching or file writing doesn't mix data. If concurrent usage is expected, consider isolating each run's results (maybe include a unique ID or timestamp in the results filename and adjust the results endpoint to read the latest or specific file). This wasn't initially handled, but could be an extension if multi-user or multi-session operation is in scope.

    Testing will also confirm adherence to coding standards – e.g., ensure that after changes, everything is still ESM-compliant (if using multer, use the ESM import syntax as allowed by its documentation or a lightweight alternative if multer is not ESM-friendly). Check that no new heavy regex or forbidden approaches slipped in. Logging should be kept informative (the console logs for each keyword result are useful for debugging and can remain, perhaps prefix them with an emoji or tag for clarity).

By implementing these fixes, the scraping tab will become fully functional and aligned with the intended design. The user will be able to upload a CSV, the system will handle it gracefully (providing feedback on any issues), DataForSEO will supply the metrics which will be reliably obtained and cached when possible, and the results will be presented in-app in a clear, styled table. This will transform the scraping tab from its

current prototype-like state to a polished feature of the MarketScout SEO Supertool, fulfilling the roadmap requirements and providing a solid user experience.

---

[1] [2] [3] Alternatives to Google Ads API for Keyword Metrics and SERP Insights.pdf
file://file-HcenvJcNWY7ceK17mBbEYR